Design and Analysis of Data Management in Scalable Parallel Scripting

Zhao Zhang*, Daniel S. Katz†, Justin M. Wozniak‡, Allan Espinosa*, Ian Foster*†‡
*Department of Computer Science, University of Chicago
†Computation Institute, University of Chicago & Argonne National Laboratory
‡Mathematics and Computer Science Division, Argonne National Laboratory

***Abstract*—We seek to enable efficient large-scale parallel execution of applications in which a shared filesystem abstraction is used to couple many tasks. Such parallel scripting (*many-task computing, MTC*) applications suffer poor performance and utilization on large parallel computers because of the volume of filesystem I/O and a lack of appropriate optimizations in the shared filesystem. Thus, we design and implement a scalable MTC data management system that uses aggregated compute node local storage for more efficient data movement strategies. We co-design the data management system with the data-aware scheduler to enable dataflow pattern identification and automatic optimization. The framework reduces the time to solution of parallel stages of an astronomy data analysis application, Montage, by 83.2% on 512 cores; decreases the time to solution of a seismology application, CyberShake, by 7.9% on 2,048 cores; and delivers BLAST performance better than mpiBLAST at various scales up to 32,768 cores, while preserving the flexibility of the original BLAST application.**

I. Introduction

Many interesting applications can be constructed naturally and easily with the scripting paradigm [14]. We focus here on many-task computing (MTC) applications [17], in which existing sequential (or parallel) programs are linked by files output by one program being used as input by others. The Montage astronomy image processing application [7] is conveniently expressed in these terms.

One approach to parallelizing a scripting application is to rewrite it as a monolithic program by using a parallel library such as MPI or a PGAS language [32]. Communications that originally occurred via filesystem operations then occur via messaging. However, this approach can be labor intensive and can lead to code maintenance problems if the original program's authors continue to evolve it in the original style. For example, while the popular mpiBLAST [10] rewrite of BLAST has good parallel scalability and high efficiency, the substantial effort required to re-engineer BLAST for MPI execution means that its latest version was built from NCBI BLAST 2.2.20, while BLAST itself has evolved to 2.2.26 as of April 2012. In contrast, substituting one version of a program for another in a script is a trivial task, unless the program's interface changes substantially.

A scripting application that involves the processing of many files can provide numerous opportunities for parallel execution, as when—to give a trivial example—we run sequential BLAST over many files. However, we can encounter problems scaling the scripting paradigm to large parallel computers because of the bottleneck inherent in having all interprocess communication occur via filesystem operations. Even if a parallel computer provides a shared filesystem accessible from all nodes, the volume and frequency of filesystem operations generated by a large scripting computation will often be overwhelming [33].

To illustrate the problem, we show in Figure 1 the core-time distribution of a Montage benchmark problem on 512 cores of an IBM BG/P with intermediate results stored on GPFS. Even on this small number of cores, I/O dominates: 73.6% of the core-time is consumed by I/O, 13.4% of the core-time is consumed by task execution; and 13.0% of the core-time is consumed by scheduling overhead and CPU idle time due to workload imbalances. Of the latter time, around 39% (5.1% of total time) is idle time due to a gather operation, in which all but one core sit idle while data is fetched from GPFS.

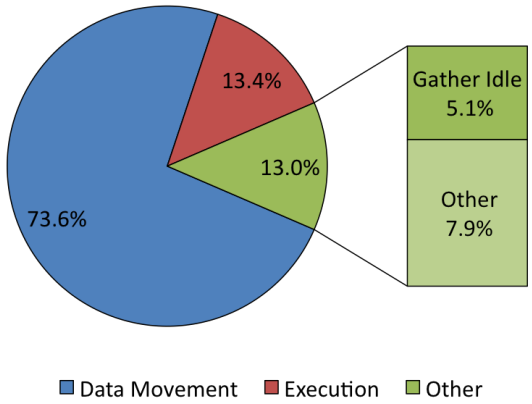

Fig. 1. Time distribution of generating a 6x6 degree mosaic with Montage on 512 IBM BG/P cores, using GPFS.

We describe here an approach to MTC intertask communication optimization based on (1) the interception of filesystem operations performed by the programs composed by an original script and (2) the implementation of the communications implied by those operations in more efficient ways. For example, we use RAM disk rather than a disk-based filesystem to hold intermediate results, and we identify and optimize specific dataflow patterns (defined in §II-A), one of which is that a single task reads (as input) all the files output by tasks in the previous stage. The result can be message-passing efficiency without program rewriting, as we will show for two different applications, Montage and BLAST.

We implement our approach within an MTC execution framework that is built upon two subsystems: an execution

SC12, November 10-16, 2012, Salt Lake
978-1-4673-0806-9/12/$31.00 ©2012 IEEE

engine AME [34] (Any-scale Many-Task computing Engine) and a data management system AMFS (Any-scale Many-Task File System). AMFS implements collective data management functionality that optimizes data movement within dataflow patterns at large scale. AME executes large-scale MTC applications, invoking AMFS functions when it detects relevant dataflow patterns. AMFS and AME together allow us to schedule tasks according to data locality at large scale, identify dataflow patterns at runtime, and apply data movement optimization automatically. As a side benefit, AME can resolve task dependencies based on file availability in a distributed manner.

This work has two major contributions. First, we introduce and demonstrate the effectiveness of automated methods for identifying, at runtime, dataflow patterns that are amenable to collective optimizations, with no need for programmer or compiler effort to identify collective operations. Second, we show that these methods plus related optimizations for caching of intermediate files can be used to execute MTC applications at high speeds on parallel computers.

As shown in Figure 2, we can characterize a filesystem by the following metrics: read and write bandwidth (bytes/sec), read and write throughput (operations/sec), locality support, and collective operations support, all normalized to being able to support 100% of applications. Today's global shared filesystems can fully address the I/O requirements of only a subset of applications. Our approach extends the filesystem interface with operations that our execution engine can use to access collective operation and file location discovery capabilities. The engine can query the filesystem for file locations through both synchronous and asynchronous interfaces, for existing files and not-yet-produced files, respectively. By providing a data cache layer between compute node RAM disk and the global shared filesystem, our mechanisms eliminate unnecessary global filesystem accesses, thus enhancing both read/write bandwidth and throughput.

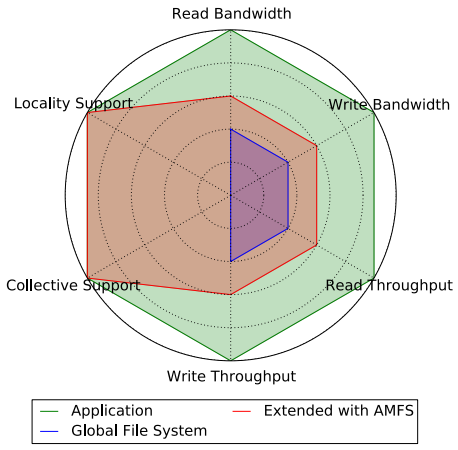

Fig. 2. Applications filesystem requirements as addressed by a global filesystem and by a filesystem extended by AMFS (qualitative).

Benchmark tests show that the collective operations built in AMFS scale to 32,768 cores and fit the model we define in §VI-A2. Application performance evaluation shows that the data management system reduces the time to solution of parallel stages of Montage by 57.3% for a 6x6 degree mosaic on 512 cores, decreases the time to solution of CyberShake by 7.9% on 2,048 cores, and provides BLAST performance comparable to that of mpiBLAST, on 256 cores to 32,768 cores.

II. Background

We introduce the dataflow patterns that we consider in this work, and we review previous work on the implementation and identification of collective operations.

A. Dataflow Patterns

Dataflow patterns have been well studied in previous papers [31], [28]. Building on our previous work [8], we define five primitive dataflow patterns, as shown in Figure 3:

- `Gather`: A small number of tasks take as input the output from a large number of tasks. Examples include a task that checks the results of previous tasks for a convergence criterion and a task that calculates summary statistics from the output of many tasks.
- `Reduce`: A small number of tasks take as input the processed output from a large number of tasks, where the processing is done by a user-specified function. The user must ensure that the function can be applied to any subset of the outputs, and the input and outputs of the function must have the same type, so that the function can be repeatedly applied. Examples include WordCount, Distributed Grep, and PageRank [2].
- `Pipeline`: A set of tasks operates on given data in sequence, with the output of one task becoming the input of the next. An example is preprocessing the input file, then applying the analysis to the preprocessed file.
- `Scatter`: A set of data must be distributed to a set of subsequent tasks. This could be the output of a previous task, or it could be data from the global filesystem. An examples is a single task extracting a group of small pieces from a large database, and distributing them to a group of subsequent tasks for further processing.
- `Multicast`: One piece of data is consumed by more than one subsequent task. An example is a database processed by a group of tasks on a set of nodes.

B. Related Work on Collective Operations

MPI [26], [23] libraries and Partitioned Global Address Space (PGAS) [32] languages introduce specialized collective operations to implement a range of dataflow patterns. They are concerned mainly with in-memory data, not files. Allthough proper interfaces can be implemented [25], [24], the collective primitives in MPI and in PGAS languages coordinate processes for collective operation only on one file at a time and are not flexible enough to handle the complex POSIX file management in MTC.

Chirp [22], a runtime shared filesystem for clusters and grids, provides limited collective-style data management.

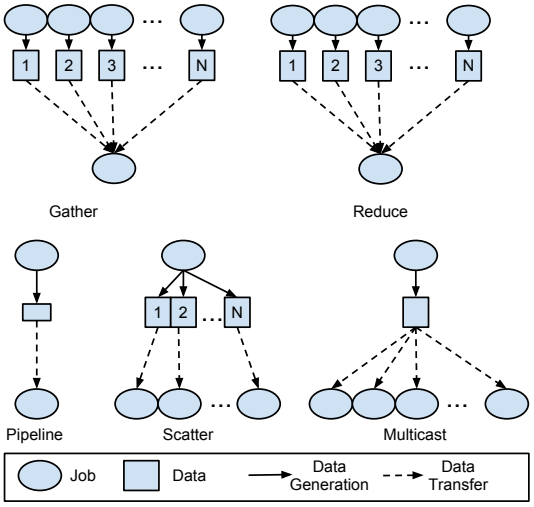

Fig. 3. Dataflow patterns.

The Chirp distribute function is identical to MPI broadcast. HDFS [1], the shared filesystem for Hadoop, replicates data a fixed number of times for fault tolerance but not according to file usage for load balancing.

Many precursors to our work on scheduling methods exist. For example, Ranganathan and Foster have conducted extensive simulation studies of data-aware scheduling methods [20], and Chakrabati et al. [3] and Desprez and Vernois [4] have studied the integration of scheduling and data replication. Our work differs in its focus on specific dataflow patterns, in particular collective patterns, and its evaluation in the context of a large, realistic application on a modern supercomputer. Work in data diffusion [19] implements data-aware scheduling-based data management and explores various algorithms for data replacement policy. This work is limited by its centralized metadata server design, and it does not provide a data management facility for various dataflow patterns of the tasks.

A preliminary version of a collective data management system [33] was previously implemented within Swift [29]. That system used RAM disks on the BG/P I/O nodes to cache data before transferring it to GPFS, in order to improve write speeds to GPFS: once intermediate data stored in cache exceeded a size limit, the system flushed the cache to GPFS. It also used the BG/P I/O nodes in order to improve the performance of broadcasting common input files to multiple tasks. However, its design was limited to the BG/P, and it required user directives to use one of these optimizations.

C. Related Work on Dataflow Pattern Identification

Dataflow patterns can be identified by the user, the compiler, or an optimizer.

User Input: MPI [26], [23] and PGAS [15] programming languages require users to explicitly declare when and where to invoke collective primitives; the user is responsible for guaranteeing the correctness of synchronization among processes. Pattern identification in [27] is based on Swift CDM (collective data management) [30]; both require users to explicitly define dataflow patterns based on a regular expression of the file names.

Compiler Analysis: Pegasus [21] uses label-based task clustering in order to optimize data locality in the pipeline dataflow pattern for intersite task scheduling. This idea can also be applied to intrasite scenarios. Nevertheless, this work employs a static approach at compile time, whereas our solution is based on runtime analysis.

Application Profiling: Preissl et al. [16] propose to identify hand crafted collective operations in MPI by profiling the execution trace. The issue solved by this work is conceptually equivalent to ours. But in our scenario, there is no explicitly annotated data transfer using send() or recv(). Instead, the data transfer is shown by a file being an input or an output.

III. Motivating Applications

We have worked with three well-known applications with MTC characteristics: Montage, BLAST, and CyberShake. Together these three applications span a range of CPU, memory, and disk I/O needs, as shown in Table I. Because of space limits, we describe only BLAST and Montage here; CyberShake is described elsewhere [11], [6].

TABLE I
Resource usage

Application	CPU	Memory	Disk I/O
Montage	Low	Low	High
CyberShake	Mid	High	Low
BLAST	High	Mid	Mid

A. Montage

Montage [7] is an astronomy application that builds mosaics from a number of small images. It has been successfully run in parallel with MPI, Pegasus [9], and Swift [18]. In the MTC version of Montage, we divide the code into eight stages, as shown in Table II.

TABLE II
Montage tasks

Stage	Description
mProject	reprojects raw images
mImgtbl	aggregates reprojected image metainfo
mOverlaps	identifies the overlapped reprojected images
mDiffFit	fits a plane to the overlapped images
mConcatFit	aggregates output of `mDiffFit`
mBgModel	produces background rectification coefficients
mBackground	applies coefficients to the reprojected images
mAdd	aggregates corrected images

B. BLAST

BLAST (the Basic Local Alignment Search Tool) searches one or more nucleotide or protein sequences against a sequence database and calculates similarities. It has been parallelized with different frameworks. For example, mpiBLAST [10] wraps BLAST as a library within an MPI framework, CloudBLAST [13] uses MapReduce as the wrapper, and Parallel BLAST [12] uses PVM as the parallel framework and maintains a POSIX file interface among stages. We start with

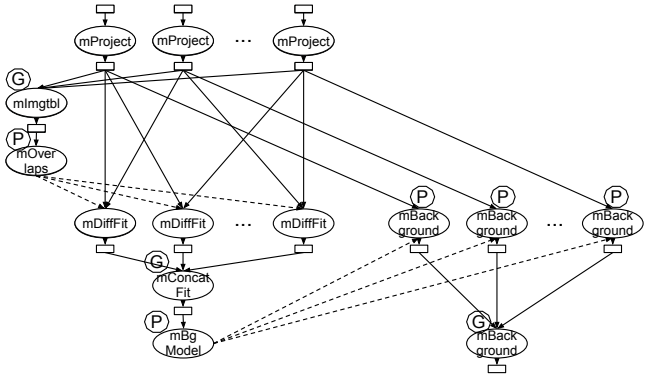

Fig. 4. Montage Dataflow Patterns: (G), Gather; (P), Pipeline.

TABLE III
BLAST TASKS

Stage	Description
fastasplit	splits the database into several slices
formatdb	formats each database slice
blastp	searches protein sequence against database slice
merge	merges the results for each query

Parallel BLAST as our base case and scale this implementation with our runtime and data management system.

Figure 5 shows the dataflow patterns in Parallel BLAST, and Table III explains the tasks in each stage.

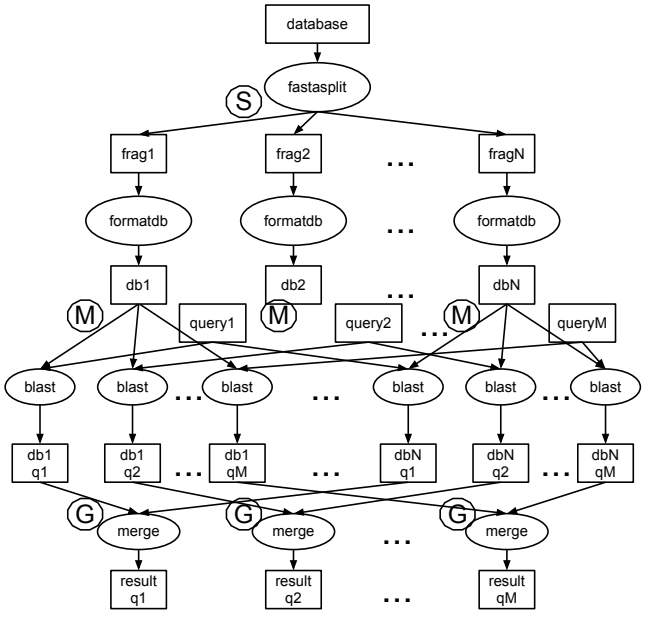

Fig. 5. Parallel BLAST Dataflow Patterns: (S), Scatter; (M), Multicast; (G), Gather.

IV. Design Alternatives

The basic idea in the AMFS data management system is to cache intermediate data in RAM disk on compute nodes in order to avoid unnecessary data movement between compute nodes and the global filesystem. We discuss here alternative solutions to the problems we found in building the system, and we evaluate them and make decisions based on these capabilities.

A. Data Movement at Large Scale

Assuming a task knows the number and location of its input files, the execution framework could fetch those input files to a single node and then start running the task. After the task completes, it has produced one or more output files. Assuming that the worker is aware of where to send those output files, a naive transfer scheme might move the files in a *sequential* manner. A straightforward improvement is to transfer the files with multiple processes in order to *parallelize* the file movement. Thakur and others have studied data movement extensively in the context of MPI [26], [23] and found that a minimum spanning tree (MST) is often an effective solution.

Figure 6 shows the runtime for the multicast, gather, and scatter patterns on varying numbers of nodes and when using sequential, parallel, and MST methods. MST is fastest in all cases studied, because all transfers considered are dominated by network latency rather than by bandwidth. We apply an asynchronous transfer (§IV-C) scheme when network bandwidth overhead dominates.

B. Dataflow Pattern Identification

We next discuss approaches to identifying dataflow patterns in MTC computations. The user could identify the patterns in the higher-level programming language through annotation, or an intelligent compiler could detect patterns at compile time. Or, the runtime could identify the dataflow patterns using the tasks it is allocated, which is the most challenging solution because the runtime might not have the insight of an application programmer or the global view of data and tasks of a compiler. We nonetheless pursue this approach in order to alleviate the burden on domain scientists and compiler writers.

C. Gather Pattern Operation

For the gather data movement pattern, we focus on two cases: (1) files that are so small that file transfer time is dominated by network latency; and (2) files that are sufficiently large that file transfer time is dominated by network bandwidth. For the first case, we use the *collective gather*, which synchronously moves all files for a gather pattern task in a MST topology. For the second, we use another gather implementation, called *asynchronous gather*, which transfers the input files for a gather pattern task based on file availability. A worker that is to gather files with the second scheme proceeds as follows. For each of the task's input files, it requests that file from AMFS. If the file is available, then the worker copies the file. Otherwise, the worker proceeds to the next input file and uses another thread to wait for acknowledgments of all the not-yet-available files. As each file is produced, the worker is notified and copies the file. We call this *asynchronous gather* in order to differentiate it from the *collective gather*, which is synchronous with the task completion of the previous stage. Collective gather benefits from the parallel transfer at each step, whereas asynchronous

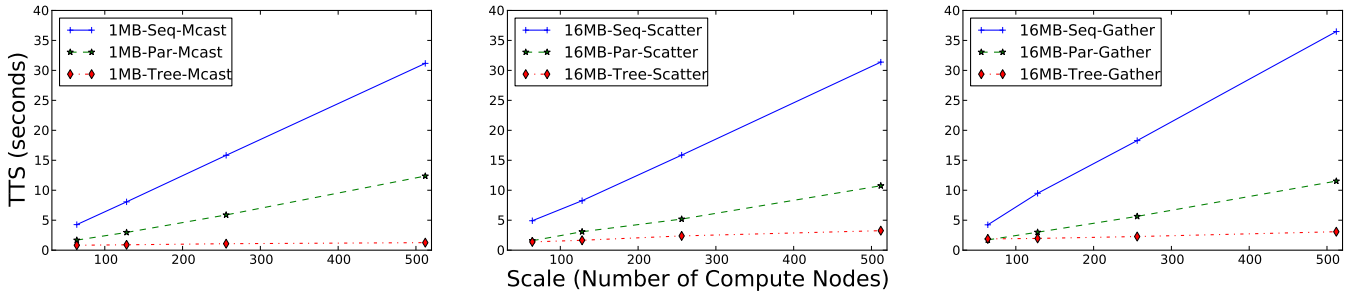

Fig. 6. Performance comparison of (from left to right) multicast, gather, and scatter dataflow patterns, using sequential, parallel, and AMFS data transfer.

gather benefits from overlapping computation and file transfer. We model the performance of the two schemes in §VI-A. We apply collective gather when tasks prior to the gather operation finish close to each other or files are small; and we apply asynchronous gather when tasks prior to the gather operation finish irregularly or files are large.

V. System Implementation

As noted above, our data management solution is built on two major components: AME, the Any-scale MTC Engine [34] and AMFS, the Any-scale MTC File System. These two components together constitute a system with the following capabilities: executing MTC application on large scale computers, identifying dataflow patterns at runtime, applying optimized collective operations in the right place, using large-scale data-aware scheduling, and resolving task dependencies in a distributed manner. Sections V-A, V-B, and V-C introduce the system architecture and the communications among the components. The rest of this section discusses technical solutions to problems that we encountered during implementation.

A. AME Architecture

The solid outlined components in Figure 7 show the AME architecture. AME has three components: a centralized submitter, a group of distributed dispatchers (located on compute nodes), and a set of workers (also located on compute nodes). The nodes with workers are divided into groups, and each group is associated with a dispatcher. The submitter reads in a Swift script or Pegasus workflow definition and, via a process of either compilation (for Swift) or translation (for Pegasus), generates a task list that it then partitions into several pieces, one for each dispatcher. The submitter places these task lists on the global shared filesystem and instructs the dispatchers to start executing. Each dispatcher then reads its assigned task list and dispatches those tasks to its workers in a round-robin manner. When a worker finishes a task, it notifies the dispatcher. When all tasks within one dispatcher have completed, the dispatcher notifies the submitter. The submitter waits for all dispatchers to return. To remedy load imbalances resulting from irregular task sizes, we implement a work-stealing algorithm among AME workers; see §V-I.

B. AMFS Architecture

The dashed outlined components in Figure 7 show the AMFS architecture. AMFS comprises a set of metadata servers

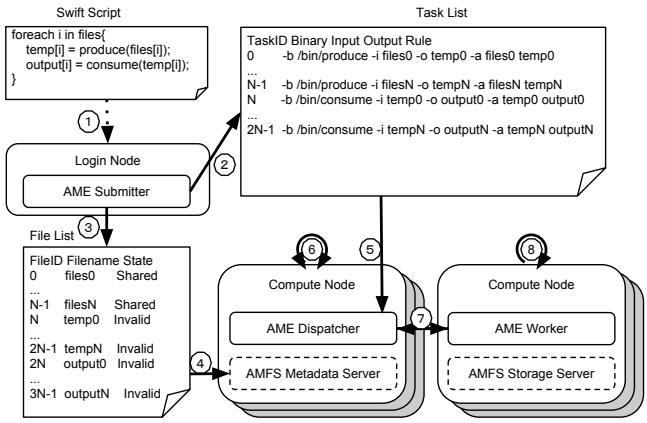

Fig. 7. Integrated AMFS/AME architecture. The dashed oblongs represent AMFS components.

and a set of storage servers, each of which runs on compute nodes. Metadata servers run on dedicated compute nodes, collocated with dispatchers, while storage servers are collocated with the AME workers just discussed. The ratio between the numbers of nodes used for metadata servers/dispatchers, and the ratio of storage servers/workers is set to 1:63 in this deployment, but is configurable.

1) Metadata Server: The metadata server implements a distributed hash table (DHT), with the file identifier as the key and the value as a compound data structure that includes the file state, location, and subscribers (the dispatcher that queues the task that takes this file as an input). The file identifier is the concatenation of the POSIX file path and the file name. A file identifier is unique in the namespace within a single application run. The metadata server supports two types of access: synchronous and asynchronous. When an AME dispatcher queries a file's location, if the file has been produced, the metadata server returns the address. Otherwise, the metadata server returns an "N/A" message and registers the dispatcher as a subscriber to that file. When the file is produced, the location will be broadcast to the subscribers.

2) Storage Server: All other compute nodes are both AME workers and AMFS storage servers. All intermediate files that are produced locally on a node are stored in its local storage. When a task completes, the AME dispatcher that originally submitted the task updates the metadata server for files that were produced by the task. The storage server includes a background FTP daemon for peer-to-peer and collective file

transfers.

3) Collective Operations: Assuming intermediate files are produced and then physically located in the local storage of a set of compute nodes, we build a collective data management toolkit that runs as a daemon (referred to here as "collective daemon") on each storage server. This toolkit optimizes the dataflow of the multicast, scatter, gather, and reduce patterns. The storage servers that are involved in a collective operation are organized into a minimum-spanning-tree topology, and the collective operation is initialized at one of storage servers by sending to its local collective daemon a message containing the pattern, the names of the files that it is expecting, and the addresses of the nodes on which those files are located. The local collective daemon that receives this message then acts as the root of the minimum spanning tree. Each collective daemon that receives an address list starts a loop until that address list is empty. In each iteration, the daemon removes half of the remaining address list (a list of children), picks the first removed address as a target, and sends the other removed addresses to the target.

In gather, a collective daemon that receives a message indicating it is a leaf makes an archive of the file(s) named within the message and acknowledges its parent. Upon receiving the acknowledgment, the gather daemon will transfer the archive from the child and check the number of children and the number of acknowledgments. If they are the same, that means all of its children returned, and the gather daemon acknowledges its parent recursively up to the root of the tree. The file flow is illustrated in Figure 8.

The root node of a reduce operation will produce the children lists similarly to gather, replacing the archiving operation with a user-defined reduce function. The nodes of a multicast operation produce the children list first, copy the file to the target of each children lists, then inform the target with the children lists. They wait for all the targets to return, then send acknowledgments to their parents. Scatter is implemented similarly to multicast, except that instead of sending the same file at each step (as in multicast), scatter makes an archive of the files that are destined for each children list and forwards this archive to the target of each children list. Upon receiving all the acknowledgments from the targets, the node acknowledges its parent.

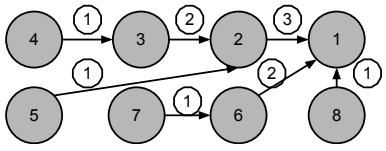

Fig. 8. File flow in collective gather on eight nodes. Shaded circles are nodes; non-shaded circles show timesteps.

C. Integrated Design of AMFS and AME

The key feature of the integrated design is the communication between AMFS and AME, shown by ④, ⑥, and ⑧ in Figure 7, which demonstrates the architecture of the framework as a whole. The AME submitter compiles (which currently supports only static workflows) a Swift script (①), and produces one file list (②) and one task list (③). The file list includes all the input, intermediate, and output files with their states. The task list includes all (or some, for a dynamic Swift program) tasks. The file list is partitioned into several pieces, and fed into the AMFS metadata servers (④). The task list is spread across the AME dispatchers (⑤). The communication between the AME dispatcher and AMFS metadata servers for data-aware scheduling is shown as ⑥. When a task is queued at an AME dispatcher, before the dispatcher sends the task to a worker, it will query the AMFS metadata server for the input file states. If the file already has been produced somewhere, the AMFS metadata server will return the address of the node. Otherwise, the AMFS metadata server will subscribe the request address to that particular file and return a message telling the dispatcher the file has not yet been produced. Then the dispatcher will put the task away temporarily and continue with other tasks. Later, when that file is produced, the AMFS metadata server will broadcast the location of that file to all the subscribers. The communication (⑥) is synchronous in the case where the file has already been produced, and asynchronous in the case of the file is invalid. Task and notification flow between AME Dispatchers and AME workers is shown by ⑦. AME workers retrieving files from other storage servers are shown by ⑧.

D. Collective Deadlock Avoidance

In our initial design, a deadlock scenario could arise with two collective operations on overlapped storage servers, when two servers concurrently consider each other as a child. We avoid this situation by predefining a global order for each storage server and requiring the files to flow only from lower order to higher.

E. Scalable Data-Aware Scheduling

Although AME and AMFS have an integrated design, their data logic and task logic are fully independent. The communication between AME and AMFS happens at two places: first, when AME compiles a Swift script to produce a file list to feed the AMFS metadata server, and second, when AME queries a file status from AMFS in both synchronous and asynchronous ways. If we have only one AMFS metadata server, it won't work in a scalable manner beyond some point, when the scale exceeds a single node's capacity (either CPU or memory). To balance load related to metadata, we choose a hash function that evenly distributes the metadata for each stage across all the metadata servers.

F. Multicast Handling

We have three subproblems to solve for multicast: (1) identify the dataflow pattern, (2) determine the number of replicas, and (3) decide where to place the replicas for load balancing. We present the details of multicast as an example of the collective operations because it is the most complicated and because it involves pattern identification as well as quantitative decision. Initially we consider a simple example, with one

TABLE IV
STAGE-INFILE-TABLE

Key	Value
a	[]
b	[file1, ..., file100]

TABLE V
SUBTABLE WITH STAGE B

Key	Value
file1	[task1, ..., task10]
file2	[task11, ..., task20]
...	...
file100	[task991, ..., task1000]

AME dispatcher, one AMFS metadata server, and 200 AMFS storage servers. Two stages of computation are involved: a and b. We have 100 tasks in stage a, and 1,000 tasks in stage b. The tasks in stage a do not have input files, and each produces one output file, which will be consumed by 10 tasks in stage b. Each stage b task has only one input file.

1) Pattern Identification: The single AME dispatcher pre-processes all the tasks and produces two hash tables. The first one has the keys being the name of the stage, which could be the application binary names. The values are a list of input file names associated with all the stage's tasks. So in our simple example, the table, shown in Table IV, has two records. The dispatcher also produces another nested table with stage name as key; the value is a subtable, with keys being the input file names and values as lists of tasks that will consume each input file, as shown in Table V. Using this table, an AME dispatcher identifies a multicast pattern by comparing the number of subsequent tasks of a file with some predefined threshold.

2) Replica Number Decision: Once a file is produced and the AME dispatcher is notified of the address of that file, we compute the fraction of this file's consumption among its peer files in the same stage. In this case, file1 will be consumed by ten stage b tasks. The consumption of file1 is 1% of the total file consumption for input files in stage b. Thus we need to multicast file1 to 1% of the AMFS storage servers, which is two. Since the file is already on one AMFS storage server, we replicate the file only once.

3) Replica Placement: To distribute the input files within a single stage in a balanced way, we need the exact AMFS storage server and data mapping. This information is stored in a table with storage server address as key and the file list as value. The table can be initialized when stage a tasks are dispatched to AME workers. In this case, the AMFS storage servers are collocated with AME workers on the same node, so we put the worker's address and output file name into the table. Then we select the AMFS storage servers with the least number of input files in this stage to hold the replicas.

4) Extending the Solution: Multiple AMFS metadata servers will not make it more difficult to solve the issues above, since each metadata server will be contacted by only one AME dispatcher in order to get file state update and location information. But having multiple AME dispatchers does raise some issues. While each AME dispatcher could still produce the tables such as Tables V and IV, some of the input files of stage b will not be produced within this dispatcher's scope. However, the dispatcher will still get an on-time file state update as long as that file is an input of some task of stage b and this specific task is in the queue of this AME dispatcher. Therefore, we need to make a small change in the above solution in order to make it work with multiple dispatchers. If the file is not produced within this dispatcher's scope, and there should be N replicas from this dispatcher's point of view, then we make N copies of that file within this dispatcher's scope. Otherwise, if the file is produced within this dispatcher's scope, the dispatcher should make N-1 replicas.

We also introduce a new concept of *file group*. A file group is defined by all input files that are also output files of the previous stage for a stage b task. We replace the role of file in the Tables V and IV with file group. The files are replicated based on the production of the whole group.

We also consider a potential problem when the number of stage b tasks is less than the number of AME workers. In this case, we simply replicate the stage b input files proportionally to their consumption.

5) Replicating Tasks: We implement an alternative solution to replicate tasks that produce intermediate files, rather than replicating the files. With the information in Tables V and IV, the system is aware of the number of desired replicas to balance the workload of the next stage of computation. Hence, it could replicate the tasks in order to produce multiple copies of the intermediate files. Then when the AME dispatchers start to send the stage b tasks, the tasks can be dispatched so that each replica gets an even number of tasks in stage b.

G. Other Collective Operations Handling

Gather is identified by either the AME dispatcher or the AME worker counting the number of input files of a task.

Reduce is identified in the same way as gather.

Pipeline is identified by default, as long as AME is capable of data-aware scheduling.

Scatter can be correctly but not efficiently identified, since the dispatcher needs to communicate with all AMFS metadata servers in order to determine the subsequent tasks. If it is a single task where the input and output file sizes exceed the compute node memory limit, we run it on the login node.

H. Resolution of Distributed Task Dependency

Task dependency resolution is a side benefit of the integrated design of AMFS and AME. The AME dispatcher quickly learns of file production by receiving file location broadcasts from the AMFS metadata server. It then knows which tasks are ready to run.

I. Load Balancing

The AME and AMFS framework uses a naive data-aware scheduling scheme to make scheduling decisions. A task with multiple input files will be dispatched to where the first input file is. Thus, a starving situation might occur, where some AME workers are busy and have extra tasks in their queues, while other AME workers are idle with empty queues. To remedy this problem in a scalable way, we implemented a simple heuristic solution: simplistic work-stealing. When a

AME worker is idle, it will try to steal a task from its neighbors. This solution is not the global optimal solution; however, it can remedy the load imbalance issue to some extent, as shown in application results in §VI-B2a.

VI. Evaluation

To evaluate our system implementation, we use models and applications. We model the performance and scalability of the AMFS collective operations and compare them with measured data. The applications we discuss in detail are Montage and mtcBLAST. We show how various combinations of techniques benefit performance. We run on an IBM BG/P, in which each compute node has four 850-MHz cores, and most communication between nodes uses a 3-D torus network, though there is also a tree network used for collective communication. I/O communication between compute nodes and GPFS uses a 10-Gbps Ethernet network.

A. Collective Operation Benchmarks

We first use a theoretical model to describe expected collective operation performance. We then compare observed performance against the models, in order to confirm the correctness of the implementation.

1) Model: We base our model for collective operations on methods frequently used in parallel computing [5], [26], [23]. In the following, `T` is the total data transfer time, `S` is the total amount of data transferred, `N` is the number of nodes, `M` is the total number of files, `a` is the latency overhead for each file transfer, and `b` is the bandwidth overhead per byte. We extend the model with `c` as overhead per file, as used in Scatter and Gather, where we put the files into an archive and then start the transfer to reduce the latency overhead of the file transfer.

- Multicast: $T = (\log_2 N) * (a + b) * S$
- Gather: $T = (\log_2 N) * a + \frac{N-1}{N} * S * b + (M-1) * c$
- Scatter: $T = (\log_2 N) * a + \frac{N-1}{N} * S * b + (M-1) * c$
- Reduce: $T = (\log_2 N) * a + S * c$

2) Performance Benchmark: Figure 9 shows the performance and scalability of the multicast, scatter, gather, and reduce dataflow patterns. In multicast, we send one file to all other nodes and vary the file size from 1 MB to 10 MB to 100 MB. In the scatter and gather tests, we fix the total amount of data that is being transferred. To benchmark the performance of reduce, we implemented a min() function as a Linux binary. This function takes a randomly generated file with integers as input and then outputs the smallest number within the file. In each test of reduce, we vary the data size on each node from 1 KB to 1 MB to 16 MB. From Figure 9, we see that multicast, scatter, gather, and reduce scale well up to 8,192 nodes. The absence of Scatter and Gather data on 8,192 nodes is due to RAM disk size limit and minimum chunk-size issues. The gap between the measured performance and the model grows larger as the number of compute node increases, because our model assumes a uniform latency between any two nodes; in reality, however, the average distance between nodes (and the latency) grows with allocation size. Also, our model does not take network congestion into account, which may also reduce effective network bandwidth.

B. Applications

We next present the results of application performance studies, analyzing the improvement contributed by each of our techniques for each application. We also compare the performance of our MTC version of Montage and BLAST with MPI versions. (We have also run the CyberShake postprocessing application, obtaining a performance improvement of 7.9%, but do not show details here, because of limited space.)

1) Montage: We built a 6x6 degree 2MASS mosaic centered at Galaxy m101. Table VI shows the number of tasks, input files, and output files and the amount of I/O performed for each stage. Figure 10 shows performance results for the MPI version of Montage and for each of the optimization methods *data cache* (CACHE), *data-aware scheduling* (AWARE), *collective gather* (COGATHER), and *asynchronous gather* (ASGATHER). In each case, we show the time-to-solution ratio for each parallel stage (except `mAdd`), the sum of the parallel stages (Sum_Para), and the sum of the whole workload (Sum_All) on 512 BG/P cores, compared with a base case, STAGING. Ratios less than 1 represent improvements, whereas ratios greater than 1 are slowdowns.

For MPI Montage, we show only the sum of the parallel stages and the sum of the whole workload, since we cannot easily separate out the other stages. In the *asynchronous gather* column, the ratio given for `mImgtbl` represents both `mProject` and `mImgtbl`, since these two stages are overlapped when using the asynchronous method and thus the time taken by each cannot be separated; similarly, the ratio given `mConcatFit` encompasses both `mDiffFit` and `mConcatFit`.

The Montage version that we use as the STAGING base case is one in which input files are initially stored on GPFS but then staged to RAM disk for reads and writes. We use STAGING performance as the base case because the different methods used to implement the MPI and MTC versions make a stage-by-stage comparison of the MPI and MTC versions infeasible. Also, the MPI version performs all reads and writes directly on GPFS, which is extremely slow: STAGING ran in 45% of the time of the MPI implementation (with all I/O directly using the shared file system).

TABLE VI
Montage stage tasks, inputs, outputs, input and output size

Stage	# Tasks	# In	# Out	In (MB)	Out (MB)
mProject	1319	1319	2638	2800	5500
mImgtbl	1	1319	1	2800	0.81
mDiffFit	3883	7766	3883	31000	3900
mConcatFit	1	3883	1	3900	0.32
mBackground	1297	1297	1297	5200	3700

a) Data Cache: Compared with the STAGING base case, Data Cache reduces the time to solution by 39.6%, 46.7%, and 1.7%, respectively, for the `mProject`, `mDiffFit`, and `mBackground` stages. The improvements result from the elimination of writes to GPFS in the three stages and the

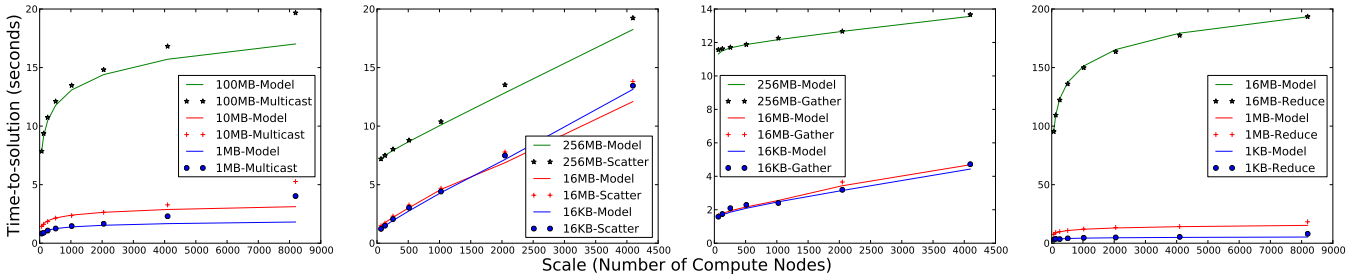

Fig. 9. Performance of the multicast, scatter, gather and reduce patterns' data movement.

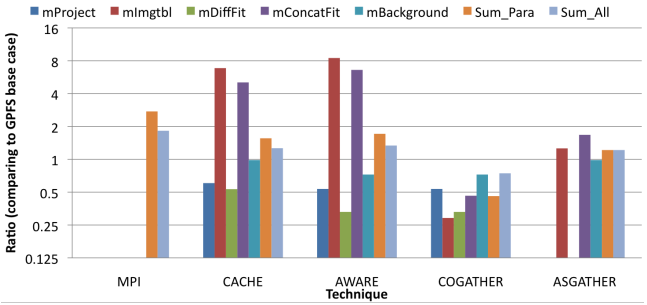

Fig. 10. Montage workload time-to-solution ratio compared with the GPFS base case for the parallel stages.

replacement of the input file copies from GPFS with peer-node file transfers in `mDiffFit` and `mBackground`. The `mBackground` output files have to be moved to GPFS in both cases, so the 1.7% decrease is due to the different input handling scheme. Overall, the Data Cache improvements are due primarily to reducing GPFS input and output, from 53.6 GB to 8.1 GB.

Stages `mImgtbl` and `mConcatFit` are 6.2x and 4.5x slower with Data Cache than STAGING, respectively. The gather pattern of these stages causes the lower performance. The `mImgtbl` and `mConcatFit` tasks have 1,319 and 3,883 input files each, and the sequential file location lookup and transfer is less efficient than file copies from GPFS.

b) Data-Aware Scheduling: Data-aware Scheduling eliminates half of the input transfer in the `mDiffFit` stage and all of the input transfer in the `mBackground` stage. In practice, data-aware scheduling reduces run time by 40.7%, and 45.1% relative to Data Cache for `mDiffFit`, and `mBackground`, and by 68.4% and 46.1% over STAGING, respectively. Stages `mImgtbl` and `mConcatFit` perform 7.7x and 5.8x slower than with STAGING, respectively, for the same reason as for Data Cache.

An unexpected improvement is that the average `mDiffFit` runtime decreases from 10 s to 1.2 s. This improvement comes from a memory cache hit for an input file, since this file is consumed on the same node where it is produced with Data-aware Scheduling and thus can be read from memory cache. We see the same result for `mBackground`, whose average runtime decreases from 8.3 s to 6.9 s.

c) Collective Gather: To optimize file transfer in the stages that have a gather dataflow pattern, we apply Collective Gather along with Data-aware Scheduling. We see a decrease in runtime of 71.0% and 53.6% for `mImgtbl` and `mConcatFit`, respectively, compared with the STAGING base case. Looking at just the file transfer time, Collective Gather saves 77.7% and 86.6% for `mImgtbl` and `mConcatFit`. The combination of these two techniques decreases the whole workload time to solution by 83.2% compared with that of the MPI Montage implementation. Table VII shows the time distribution of `mImgtbl` and `mConcatFit` with Collective Gather. In this table, input is the end-to-end gather time; execute is the execution time for the task; output is the time to write the output (short as there is only one output file per task); and overhead is system overhead. The system overheads for `mImgtbl` and `mConcatFit`, 0.80 s and 1.33 s, respectively, are due to the need to transfer and process the long task descriptions that enumerates all input files; these descriptions are 21.3 KB and 147.92 KB in size, respectively. Our alternative Asynchronous Gather solution does not perform as well as Collective Gather in this case, since the task runtime is not irregular enough to mask the data movement overhead.

TABLE VII
TIME (S) DISTRIBUTION IN GATHER PATTERN

Stage	Input	Execute	Output	Overhead
mImgtbl (GPFS)	14.74	0.43	0.07	0.55
mConcatFit (GPFS)	44.64	28.49	0.1	1.14
mImgtbl (Coll. Gather)	3.29	0.42	0.07	0.80
mConcatFit (Coll. Gather)	5.97	27.08	0.11	1.33

2) BLAST: Our parallel BLAST (which we refer to as mtcBLAST) is based on Parallel BLAST [12]. In our experiments, we match a random set of sequences selected from *nr*, a non-redundant protein database maintained at NCBI, against the full *nr* database. On n (256, 1024, 4096, 16384, and 32768) cores, we randomly choose n sequences from *nr*. We aggregate these sequences into $n/16$ query files, each containing 16 sequences; thus, our scheduling granularity is 16 sequences. Table VIII shows the number of tasks, the number of distinct inputs and outputs, and the total input and output sizes. N is defined as the number of fragments of the *nr* database, while M is defined as the number of query files, each of which contains 16 sequences.

Normally, users simply run `fastasplitn` and `formatdb` once and reuse the formatted database slices. We also run `fastasplitn` once, since it just does file fragmentation based on text; but we put `formatdb` in our mtcBLAST workload because it is an intriguing example of the multicast dataflow pattern and we can run it in parallel. We partition the *nr*

TABLE VIII
BLAST STAGE TASKS, INPUTS, OUTPUTS, AND INPUT AND OUTPUT SIZE

Stage	# Tasks	# In	# Out	In (MB)	Out (MB)
`fastasplitn`	1	1	N	4039	4039
`formatdb`	N	N	3N	4039	4400
`blastp`	N*M	N+M	N*M	73*N*M	2.4*N*M
`merge`	M	N*M	M	2.4*N*M	4.8*M

database into 63 slices; formatting those slices on 63 nodes takes 56 s. The number 63 is chosen because we dedicate one compute node out of every 64 as the AME dispatcher as well as the AMFS metadata server; the other 63 nodes are AME workers.

a) Load Balancing: BLAST task lengths are not related to the query sequence length [10], and we saw several starving situations in our small-scale test on 256 cores. Therefore, we apply a simple work-stealing mechanism in the AME worker. For scalability, we limit stealing to a compute node's neighbors (on BG/P, we choose neighbors on the 3D torus network). While this limited stealing is not a globally optimal solution, it delivered a 3% to 9% decrease in time to solution for the `blastp` stage, as shown in Figure 11.

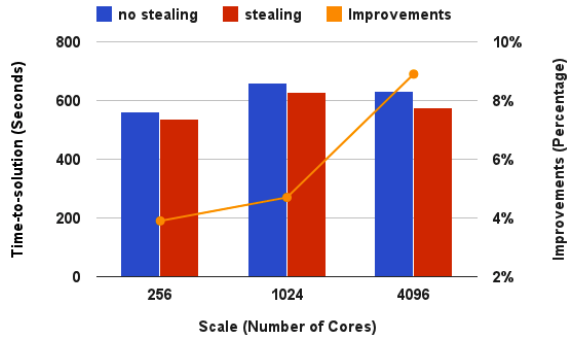

Fig. 11. Work-stealing improvement in mtcBLAST's `blastp`.

b) Asynchronous Gather: To evaluate the choice between Collective Gather and Asynchronous Gather, we run each on the same query workload. Collective Gather takes 143.3 s to finish the `blastp` and `merge` stages, while Asynchronous Gather takes 139.6 s, because of overlapping the computation of `blastp` and the input data transfer of `merge`. We therefore choose Asynchronous Gather in mtcBLAST.

c) Comparison with mpiBLAST: We compare mtcBLAST with mpiBLAST on 256 to 32,768 cores, with the same queries run against the *nr* database in both cases. For a fair comparison, mtcBLAST uses the NCBI legacy engine. Figure 12 shows that mtcBLAST is somewhat faster than mpiBLAST at each scale. This improvement is due to AME/AMFS support for large-scale data-aware scheduling, data replication, and gather dataflow optimization.

VII. Conclusions and Future Work

We have shown that an intelligent runtime data management system can significantly improve the performance of many-task applications in situations in which shared filesystem access dominates execution time. Our methods as implemented

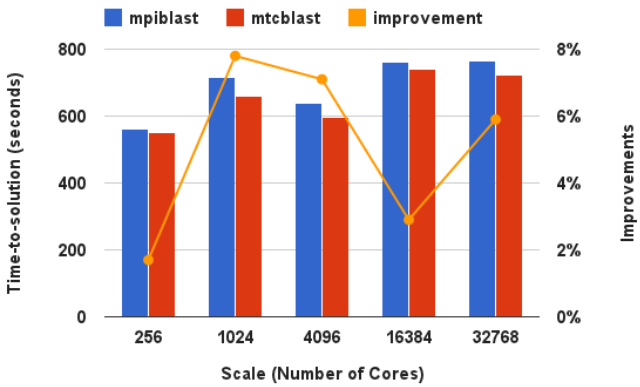

Fig. 12. Performance comparison of mtcBLAST and mpiBLAST

in AME and AMFS reduce the time to solution for parallel stages in Montage by a factor of 83.2% and reduce the time to solution of CyberShake postprocessing by 8%. Our mtcBLAST achieves performance comparable to that of mpiBLAST while preserving the flexibility of the core NCBI BLAST routines.

We also have described how tasks that feature multicast, gather, pipeline, and reduce dataflow patterns can be identified in our system at runtime, either by task analysis or by counting the number of input files. Subsequent operations and optimizations can then be applied automatically to the dataflow pattern; our system design does not require explicit user input, such as explicit procedure calls or file hints. We show examples in Montage and mtcBLAST, where either Collective Gather or Asynchronous Gather performs better; identifying the better gather techniques automatically at runtime is future work.

Inspired by the mtcBLAST and mpiBLAST comparison, we plan to characterize the I/O requirements of a group of parallel applications, identify the I/O characteristics of a set of parallel computers, and quantitatively study how an MTC middleware can extend the range of applications that can be supported on those computers. In this way, we aim to identify those applications that fit well in the MTC application category and to categorize MTC applications into those that would benefit from a proper MTC middleware and those that would not. This work will help scientists make decisions about selecting a parallel framework (MTC, MPI, etc.) for new applications based on I/O characteristics.

Acknowledgments

This work was supported in part by the U.S. Department of Energy under the ASCR X-Stack program (contract DE-SC0005380) and contract DE-AC02-06CH11357. Computing resources were provided by the Argonne Leadership Computing Facility. We thank Dr. David Mathog (Caltech) for his support with parallel BLAST, and the ALCF support team. Work by Katz was supported by the National Science Foundation while working at the Foundation. Any opinion, finding, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] D. Borthakur. HDFS architecture. http://hadoop.apache.org/hdfs/docs/current/hdfs_design.pdf.

[2] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[3] A. Chakrabarti, R. Dheepak, and S. Sengupta. Integration of scheduling and replication in data grids. In L. Bougé and V. Prasanna, editors, *High Performance Computing - HiPC 2004*, volume 3296 of *Lect. Notes in Comp. Sci.*, pages 85–101. Springer, 2005.

[4] F. Desprez and A. Vernois. Simultaneous scheduling of replication and computation for data-intensive applications on the grid. *J. of Grid Comp.*, 4:19–31, 2006.

[5] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Boston, MA, 1995.

[6] R. Graves, T. Jordan, S. Callaghan, E. Deelman, E. Field, G. Juve, C. Kesselman, P. Maechling, G. Mehta, K. Milner, D. Okaya, P. Small, and K. Vahi. CyberShake: A physics-based seismic hazard model for southern California. *Pure and Applied Geophysics*, Online Fir:1–15, May 2010.

[7] J. C. Jacob, D. S. Katz, G. B. Berriman, J. C. Good, A. C. Laity, E. Deelman, C. Kesselman, G. Singh, M.-H. Su, T. A. Prince, and R. Williams. Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking. *Intl. J. of Comp. Sci. and Eng.*, 4(2):73–87, 2009.

[8] D. S. Katz, T. Armstrong, Z. Zhang, M. Wilde, and J. Wozniak. Many task computing and Blue Waters. Technical Report CI-TR-13-0911, Computation Institute, University of Chicago, November 2011.

[9] D. S. Katz, J. C. Jacob, G. B. Berriman, J. Good, A. C. Laity, E. Deelman, C. Kesselman, and G. Singh. A comparison of two methods for building astronomical image mosaics on a grid. In *Proc. 2005 Intl. Conf. on Par. Proc. Works.*, pages 85–94, 2005.

[10] H. Lin, P. Balaji, R. Poole, C. Sosa, X. Ma, and W. Feng. Massively parallel genomic sequence search on the Blue Gene/P architecture. In *Proc. IEEE/ACM Supercomputing 2008*, November 2008.

[11] P. Maechling, E. Deelman, L. Zhao, R. Graves, G. Mehta, N. Gupta, J. Mehringer, C. Kesselman, S. Callaghān, D. Okaya, H. Francoeur, V. Gupta, Y. Cui, K. Vahi, T. Jordan, and E. Field. SCEC CyberShake workflows – automating probabilistic seismic hazard analysis calculations. In I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields, editors, *Workflows for e-Science*, pages 143–163. Springer, 2007.

[12] D. R. Mathog. Parallel BLAST on split databases. *Bioinformatics*, 19(14):1865–1866, 2003.

[13] A. Matsunaga, M. Tsugawa, and J. Fortes. CloudBLAST: Combining MapReduce and virtualization on distributed resources for bioinformatics applications. In *Fourth IEEE International Conference on eScience (eScience '08)*, pages 222 –229, dec. 2008.

[14] J. Ousterhout. Scripting: Higher level programming for the 21st Century. *IEEE Computer*, 31:23–30, 1998.

[15] Partitioned Global Address Space languages. http://pgas.org.

[16] R. Preissl, M. Schulz, D. Kranzlmüller, B. R. de Supinski, and D. J. Quinlan. Transforming MPI source code based on communication patterns. *Future Gener. Comput. Syst.*, 26(1):147–154, Jan. 2010.

[17] I. Raicu, I. Foster, and Y. Zhao. Many-task computing for grids and supercomputers. In *Proc. of Many-Task Comp. on Grids and Supercomputers, 2008*, 2008.

[18] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde. Falkon: a fast and light-weight task execution framework for grid environments. In *SC'2007*, 2007.

[19] I. Raicu, Y. Zhao, I. T. Foster, and A. Szalay. Accelerating large-scale data exploration through data diffusion. In *Proc. of 2008 Intl. Work. on Data-Aware Dist. Comp. (DADC '08)*, pages 9–18. ACM, 2008.

[20] K. Ranganathan and I. Foster. Simulation studies of computation and data scheduling algorithms for data grids. *J. of Grid Comp.*, 1(1):53–62, 2003.

[21] G. Singh, M.-H. Su, K. Vahi, E. Deelman, B. Berriman, J. Good, D. S. Katz, and G. Mehta. Workflow task clustering for best effort systems with pegasus. In *15th ACM Mardi Gras Conference: From lightweight mash-ups to lambda grids: Understanding the spectrum of distributed computing requirements, applications, tools, infrastructures, interoperability, and the incremental adoption of key capabilities (MG'08)*, pages 9:1–9:8, New York, 2008. ACM.

[22] D. Thain, C. Moretti, and J. Hemmes. Chirp: a practical global filesystem for cluster and grid computing. *J. of Grid Comp.*, 7(1):51–72, 2009.

[23] R. Thakur and W. Gropp. Improving the performance of collective operations in MPICH. In J. Dongarra, D. Laforenza, and S. Orlando, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2840 of *Lect. Notes in Comp. Sci.*, pages 257–267. Springer, 2003.

[24] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective I/O in ROMIO. *Symp. on Frontiers of Massively Par. Proc.*, page 182, 1999.

[25] R. Thakur, W. Gropp, and E. Lusk. On implementing MPI-IO portably and with high performance. In *Proc. of 6th Work. on I/O in Par. and Dist. Systems (IOPADS'99)*, pages 23–32. ACM, 1999.

[26] R. Thakur and R. Rabenseifner. Optimization of collective communication operations in MPICH. *Intl. J. of High Perf. Comp. Applications*, 19:49–66, 2005.

[27] E. Vairavanathan, S. Al-Kiswany, L. Costa, M. Ripeanu, Z. Zhang, D. S. Katz, and M. Wilde. A workflow-aware storage system: An opportunity study. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2012.

[28] M. Wilde, I. Foster, K. Iskra, P. Beckman, Z. Zhang, A. Espinosa, M. Hategan, B. Clifford, and I. Raicu. Parallel scripting for applications at the petascale and beyond. *Computer*, 42:50–60, 2009.

[29] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster. Swift: A language for distributed parallel scripting. *Par. Comp.*, pages 633–652, September 2011.

[30] J. Wozniak and M. Wilde. http://www.ci.uchicago.edu/swift/guides/release-0.93/userguide/userguide.html#_collective_data_management.

[31] J. M. Wozniak and M. Wilde. Case studies in storage access by loosely coupled petascale applications. In *Proc. 4th Annual Work. on Petascale Data Storage*, pages 16–20, 2009.

[32] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, M. Welcome, and T. Wen. Productivity and performance using partitioned global address space languages. In *2007 International Workshop on Parallel Symbolic Computation (PASCO '07)*, pages 24–32, New York, 2007. ACM.

[33] Z. Zhang, A. Espinosa, K. Iskra, I. Raicu, I. Foster, and M. Wilde. Design and evaluation of a collective I/O model for loosely coupled petascale programming. In *Proc. of Many-Task Comp. on Grids and Supercomputers, 2008*, 2008.

[34] Z. Zhang, D. S. Katz, M. Ripeanu, M. Wilde, and I. Foster. AME: An anyscale many-task computing engine. In *6th Work. on Workflows in Support of Large-Scale Science (WORKS11)*, 2011.